Venable Ref. No. 42339-198432
Intel Ref. No. P18144 (33866)

APPLICATION FOR UNITED STATES LETTERS PATENT

INVENTORS:
Jinzhan Peng
Beijing, China

Gansha Wu
Beijing, China

Guei-Yuan Lueh
San Jose, California

TITLE: Stack Caching Using Code Sharing

ASSIGNEE:
Intel Corporation
Santa Clara, California

ATTORNEYS/ AGENTS:
Venable LLP
Box 34385
Washington, DC 20043-9998
Telephone: (202) 344-4000
Facsimile: (202) 344-8300

ATTORNEY DOCKET NO.: **42339-198432**

Stack Caching Using Code Sharing

Background of the Invention

[0001] Interpretation is one of the broadly used technologies to implement Virtual Machine (VM) and runtime systems, with the benefits of portability and maintainability. VMs, runtime systems, and other high level language processors, such as Java Processors, incorporate a stack caching scheme to virtually map bytecode, for example, to an operand stack. One type of stack caching scheme, a mixed stack, may use physical registers and a contiguous memory region as an operand stack. An interpreter plays an important role in many runtime systems. Many modern programming languages, such as Java, Forth, Perl, and Python are still employing various interpreters as their execution engines when they are programmed and run on memory/computation constraint devices, for example.

[0002] The interpretation of stack-based languages may rely on an auxiliary data structure, e.g., operand stack, on which the executions of instructions are operated. Accessing the operand stack may involve memory accesses. In various methods to improve the performance of interpretation. Among those methods, stack caching may be an efficient approach to eliminate most of the accesses to the operand stack and is able to speedup interpretation. Stack caching may promote top-of-stack operands to registers, which may reduce the number of memory accesses and results in higher instructions per cycle.

[0003] To manipulate different states of a mixed stack, for example, a stack-caching interpreter may maintain many copies of execution code for each VM instruction. Such a design incurs code explosion which may consume excessive memory and introduce maintenance complexity.

Brief Description of the Drawings

[0004] Various exemplary features and advantages of embodiments of the invention will be apparent from the following, more particular description of exemplary embodiments of the present invention, as illustrated in the accompanying drawings wherein like reference numbers generally indicate identical, functionally similar, and/or structurally similar elements.

[0005] Figure 1 depicts an exemplary embodiment of a mixed stack according to an embodiment of the invention;

[0006] Figure 2 depicts an exemplary embodiment of a system according to an embodiment of the invention;

[0007] Figure 3A depicts an exemplary embodiment of a method according to an embodiment of the invention;

[0008] Figure 3B depicts an exemplary embodiment of a method according to an embodiment of the invention;

[0009] Figure 4 depicts an exemplary embodiment of a method according to an embodiment of the invention;

[00010] Figure 5 depicts an exemplary embodiment of a code layout according to an embodiment of the invention;

[00011] Figure 6A depicts an exemplary embodiment of a code layout according to an embodiment of the invention;

[00012] Figure 6B depicts an exemplary embodiment of a code layout according to an embodiment of the invention;

[00013] Figure 6C depicts an exemplary embodiment of a code layout according to an embodiment of the invention;

[00014] Figure 6D depicts an exemplary embodiment of a code layout according to an embodiment of the invention;

[00015] Figure 6E depicts an exemplary embodiment of a code layout according to an embodiment of the invention;

[00016] Figure 6F depicts an exemplary embodiment of a code layout according to an embodiment of the invention;

[00017] Figure 6G depicts an exemplary embodiment of a code layout according to an embodiment of the invention;

[00018] Figure 7 depicts an exemplary embodiment of a computer and/or communications system as can be used for several components in an embodiment of the invention.

Detailed Description of Exemplary Embodiments of the Present Invention

[00019] Exemplary embodiments of the invention are discussed in detail below. While specific exemplary embodiments are discussed, it should be understood that this is done for illustration purposes only. A person skilled in the relevant art will recognize that other components and configurations may be used without parting from the spirit and scope of the invention.

[00020] Embodiments of the present invention may provide a code sharing mechanism for stack caching that avoids code duplication. A stack caching scheme may use a mixed register-stack model, i.e. a mixed stack, that virtually maps to a bytecode (e.g., Java or CLI) operand stack. The mixed stack may consist of two parts: a register stack and a memory stack. The register stack may be comprised of physical registers that may hold several top elements of the operand stack. The memory stack may be a contiguous memory region that may hold the rest of elements of the operand stack.

[00021] Figure 1 depicts an exemplary embodiment of a mixed stack 100. As shown in Figure 1, mixed stack 100 may include a register stack 101 and a memory stack 102.

[00022] Figure 2 depicts an exemplary embodiment of a virtual machine architecture 200. Virtual machine architecture 200 may include interpreter 201, loader 202, garbage collector 203, thread 204, and native module 205. In an exemplary embodiment of the invention, interpreter 201 may include an arithmetic logic unit (ALU) (not shown), a stack (not shown), and memory (not shown). Interpreter 201 may use the aforementioned components to decode instructions and call appropriate functional units to carry out instructions. Loader

202 may be responsible for loading class files into memory, parsing the class files, and preparing bytecode instructions for interpreter 201. Interpreter 201 may be the execution engine of a VM, and may interpret instructions one at a time, for example. Garbage collector 203 may allocate new objects and reclaim useless objects. In an exemplary embodiment of the invention, thread 204 may support an application programming interface (API) and native module 205 may support the API for native library functions, for example.

[00023] In an exemplary embodiment of the invention, machine instructions may take operands from an operand stack, operate on them, and return results to the stack. A stack may be a 32-bit stack, for example, that may be used to pass parameters to methods and receive method results, as well as to supply parameters for operations and save operation results. In an exemplary embodiment of the invention, a stack may be a mixed stack as is described above.

[00024] In an exemplary embodiment of the invention, an interpreter, such as interpreter 201 may keep most, if not all, bytecode instructions to be operated on in a register stack instead of a memory stack. Doing so may reduce memory accesses and execution time of the instruction.

[00025] In an exemplary embodiment of the invention, the interpreter may need to perform shift operations to maintain the top-of-stack elements of the operand stack in the register stack. For example, if one instruction consumes the one register as is shown in Figure 3A, the top-of-stack register R1 may be removed from the register stack, the resulting value of R2 may need to be shifted to the top, R3 to R2. Because the memory stack is not empty (register stack underflow), the value in slot 1 may also be shifted to R3 so as to keep the register stack fully loaded with values. The memory stack pointer (sp) may also be updated to sp' after slot1 is drained. On the other hand, as is shown in Figure 3B, if one instruction generates one value that needs to be pushed onto the register stack, R1 may need to be vacated by shifting values down, i.e., R3 to slot 0 (register stack overflow), R2 to R3 and then R1 to R2.

[00026] While embodiments of the present invention are described in terms of the examples below, this description is for convenience only and is not intended to limit its application. In fact, after reading the following description, it will be apparent to one skilled in the relevant art(s) how to implement the following invention in alternative embodiments (e.g., in a Common Language Runtime Environment).

[00027] Furthermore, while the following description focuses interpreting JAVA bytecode, it is not intended to limit the application of the present invention. It will be apparent to one skilled in the relevant art how to implement the following invention, where appropriate, in alternative embodiments. For example, embodiments of the present invention may be applied, alone or in combination, with various virtual machine architectures, such as, but not limited to, Common Language Infrastructure and other virtual execution systems.

[00028] Figure 4 illustrates how an interpreter may work with a mixed stack in an exemplary embodiment of the invention. In an exemplary embodiment of the invention, an instruction, such as a bytecode instruction may undergo a stack-state aware translation into threaded code, which may indicate an entry point into shared execution code for executing the instruction.

[00029] Figure 4 depicts an exemplary embodiment of a transition 400 of an instruction from bytecode, for example, to shared execution code. As shown in Figure 4, prior to interpreting a method the first time, a bytecode instruction 401 may be passed to or interpreted by a stack-state-aware translator 402. The stack-state-aware translator 402 may produce threaded code 403. Based on the threaded code 403, the instruction 401 may be dispatched according to the operand stack state of the instruction. In an exemplary embodiment of the invention, when an instruction is dispatched according the operand stack state of the instruction, the entry point into shared execution code 404 may be determined and the instruction may be executed from that entry point.

[00030] In an exemplary embodiment of the invention, the stack state may be embodied by the number of shift operations that are needed after the execution of

the instruction. As used herein, η(i) denotes the number of shift operations that are needed after the execution of instruction i. For example, referring to Figure 3B, there are three shift operations in Figure 3B.

[00031] To illustrate the method as described with respect to Figure 4, the integer add instruction, `iadd`, may be used as an example to explain exemplary embodiments of the code-sharing mechanism. In an exemplary embodiment of the invention, a register stack may consist of 2 registers, for example, that include a top-of-stack (tos) register and a next-top-of-stack (nos) register. For the instruction `iadd`, there may be two possible stack states, depending on η(`iadd`), for executing `iadd`. The IADD_S1 on line 1 of execution code 404 in Figure 4 may represent the case of interpreting IADD when η(iadd) = 1 and IADD_S0 on line 3 in execution code 404 may correspond to the case with η(`iadd`) = 0.

[00032] In considering the case η(`iadd`) = 1, η(`iadd`) = 1 may occur when an operand stack has more than two elements. In other words, η(`iadd`) = 1 may occur when the memory stack is not empty and the register stack is full. Because the instruction `iadd` consumes two operands, (i.e., tos and nos respectively) and produces one (new tos), there may only be one shift operation required to move the top item on the memory stack to the register stack as the new nos. The `iadd` instruction may then be dispatched to line 1 of the IADD_S1 case (as shown in Figure 4). Line 2 of the IADD_S1 case may pop the top element of the memory stack to a temp register, for example. The execution may then fall through to the IADD_S0 case, in which the register-wise add operation (line 4) may interpret the integer add operation. Line 5 may refill nos by moving temp to nos to keep the top two elements of the operand stack in registers, for example. As described herein, the combination of lines 2 and 5 may constitute the shift operation.

[00033] In considering the case η(`iadd`) = 0, η(`iadd`) = 0 may occur when the operand stack has only two elements (both are in the register stack). In such a case, no shift operation may be needed because there may only be one element left as the result of the add operation. `iadd` may be dispatched to IADD_S0 (tos will

be the only stack item after execution). As described above, line 4 may interpret the integer add operation. Execution of the refilling statement (line 5) may then become useless and redundant, but may not affect the correctness of the program because only tos may be a legitimate item after execution of IADD. In such a case, the performance penalty may be trivial for the nature of register-register assignment.

[00034] As is shown and described, IADD_S0 and IADD_S1 may share the same execution code to avoid excessive code duplication. In an exemplary embodiment of the invention, execution code and instruction dispatching for various stack states may be reused with a comprehensively designed layout. In such an embodiment, during the code-threading phase, the stack state for each instruction may be inferred, and then the instruction may be directly dispatched to the appropriate execution entry without a runtime table lookup, for example. Additionally, the translation phase may perform some optimizations to improve the sequence of interpretation.

[00035] Figure 5 depicts an exemplary code layout 500 according to an exemplary embodiment of the invention. As shown in Figure 5, OP_S_k denotes the interpretation entry point for an instruction that has an opcode OP and needs k ($\eta(i) = k$) shift operations to maintain the operand stack after the execution of i. The general code layout of all VM instructions may be illustrated as is shown in Figure 5, for example.

[00036] In Figure 5, SO_k is the code that corresponds to the shift operation for OP_S_k. In $SO_{k,}$ the shifted elements may be moved to the register stack (RO) after execution of the operation. In an exemplary embodiment of the invention, $OP\_\ S_k$ may also execute all the code of its subsequent entries, $OP\_\ S_0$ to $OP\_\ S_{k-1}$. That is, in such an embodiment, the code of $OP\_\ S_0$ to $OP\_\ S_{k-1}$ may be shared. ID is the code that calls the next instruction.

[00037] As an example, consider the case of register stack size M = 2 (i.e., there are 2 registers in the register stack as described above). The property of an instruction i may be defined as [X(i), Y(i)], where X(i) denotes the number of

operands that i consumes and $Y(i)$ denotes the number of stack items that i produces. Figures 6A-6G enumerate all possible code layouts for of $0 \leq X(i) \leq M$ and $0 \leq Y(i) \leq M$. As an example, the previous `iadd` example falls into the category of Figure 6D.

[00038] For the instructions whose $X(i) > M$ or $Y(i) > M$, a similar style of code layout may still be applied. In an exemplary embodiment of the invention, when $X(i) > M$ or $Y(i) > M$, more register-memory shift operations may need to be performed before the execution code.

[00039] As is shown in Figures 6A-6G, each code layout represents a particular category $[X(i), Y(i)]$, where $X(i)$ denotes the number of operands that i consumes and $Y(i)$ denotes the number of stack items that i produces. In an exemplary embodiment of the invention, the stack-state-aware translation phase may complement the code layout design. In such an embodiment, the stack-state-aware translation may happen before the instruction is executed. The translator may walk through the bytecode of the instruction in a pseudo-execution manner, for example, and generate the appropriate threaded bytecode entry for each instruction. At each execution point, the translator may be aware of the operand stack state and $[X(i), Y(i)]$ property of the current instruction i. Accordingly the translator may infer $\eta(i)$ based on a static table lookup or on a calculation result of a comprehensive formula, such as $f(\mathrm{Depth}(\mathrm{opstack}), M, X(i), Y(i))$.

[00040] In the described embodiments, the correctness of the stack-state-aware translation may be based on the fact that the stack depth before and after each bytecode instruction can be determined statically (runtime invariant). Such translation may only need one pass for a majority of bytecode instructions. Such embodiments may enable more optimization opportunities that are exposed during the translation.

[00041] Figure 7 depicts an exemplary embodiment of a computer and/or communications system as may be used to incorporate several components of the system in an exemplary embodiment of the present invention. Figure 7 depicts an exemplary embodiment of a computer 700 as may be used for several computing

devices in exemplary embodiments of the present invention. Computer 700 may include, but is not limited to: e.g., any computer device, or communications device including, e.g., a personal computer (PC), a workstation, a mobile device, a phone, a handheld PC, a personal digital assistant (PDA), a thin client, a fat client, an network appliance, an Internet browser, a paging, or alert device, a television, an interactive television, a receiver, a tuner, a high definition (HD) television, an HD receiver, a video-on-demand (VOD) system, a server, or other device.

[00042] Computer 700, in an exemplary embodiment, may comprise a central processing unit (CPU) or processor 704, which may be coupled to a bus 702. Processor 704 may, e.g., access main memory 706 via bus 702. Computer 700 may be coupled to an Input/Output (I/O) subsystem such as, e.g., a network interface card (NIC) 722, or a modem 724 for access to network 726. Computer 700 may also be coupled to a secondary memory 708 directly via bus 702, or via main memory 706, for example. Secondary memory 708 may include, e.g., a disk storage unit 710 or other storage medium. Exemplary disk storage units 710 may include, but are not limited to, a magnetic storage device such as, e.g., a hard disk, an optical storage device such as, e.g., a write once read many (WORM) drive, or a compact disc (CD), or a magneto optical device. Another type of secondary memory 708 may include a removable disk storage device 712, which can be used in conjunction with a removable storage medium 714, such as, e.g. a CD-ROM, or a floppy diskette. In general, the disk storage unit 710 may store an application program for operating the computer system referred to commonly as an operating system. The disk storage unit 710 may also store documents of a database (not shown). The computer 700 may interact with the I/O subsystems and disk storage unit 710 via bus 702. The bus 702 may also be coupled to a display 720 for output, and input devices such as, but not limited to, a keyboard 718 and a mouse or other pointing/selection device 716.

[00043] The embodiments illustrated and discussed in this specification are intended only to teach those skilled in the art various ways known to the inventors to make and use the invention. Nothing in this specification should be considered

as limiting the scope of the present invention. All examples presented are representative and non-limiting. The above-described embodiments of the invention may be modified or varied, without departing from the invention, as appreciated by those skilled in the art in light of the above teachings. It is therefore to be understood that the invention may be practiced otherwise than as specifically described.